VIA C3 Processor Alternate Instruction Set

Application Note

Version 0.24

(Review version, Incomplete)

VIA Confidential

This is **Version 0.24** of the VIA C3 Processor Alternate Instruction Set App Note.

Table of Contents

CHAPTER

1

INTRODUCTION

This document describes the VIA C3 processor capability to execute an alternate set of instructions. A separate document, the
describes the details of the micro-operations that may be used as alternate instructions.

1.1 BASIC CONCEPTS

The VIA C3 processor family is intended as a plug-replaceable, software-compatible alternative to the Intel Pentium III processor. Accordingly, the VIA C3 processor normally executes compatible instructions. The internal design of the VIA C3 processor, however, is quite different from the Pentium III internal design. In particular, the VIA C3 processor comprises two major components: a front-end that fetches x86 instruction bytes and translates them from x86 into micro-operations, and an internal microprocessor that executes these micro-operations.

This Application Note discloses an additional capability: the VIA C3 processor has a special mode where the front-end logic can also fetch (selected) micro-operations (versus x86 instructions) and pass them directly to the internal execution unit. The specific micro-operations that can be fetched from memory in this fashion are called . If certain rules are carefully followed, these new instructions can be intermixed seamlessly with x86 instructions in almost any combination.

1.2 RATIONALE

The VIA C3 processor family provides the alternate instruction capability for three basic uses: processor testing, processor debugging, and selected software performance improvement.

1.2.1 PROCESSOR TESTING

Since the internal microprocessor of the VIA C3 processor family is considerably different from x86 architecture, it is difficult to test internal microprocessor features from the x86 architecture. For example, the internal microprocessor has 32 general registers that are all used in emulating the x86 architecture. It would take x86 instructions to guarantee adequate testing of all 32 internal registers, assuming that a deterministic map of x86 instructions to internal registers could be accurately produced.

By executing alternate instructions from memory, however, it is easy to directly test all of the internal microprocessor hardware features using a small number of alternate instructions. This capability is extensively used in the VIA C3 processor's production manufacturing tests.

1.2.2 PROCESSOR DEBUG

Similarly, in debugging software on the VIA C3 processor, it is often desirable to directly understand or manipulate internal microprocessor state. The alternate instruction capability allows alternate instructions to be embedded within x86 instructions to setup special states, sample internal values, and so forth. This capability is used by the Centaur Technology development and debug team.

1.2.3 SOFTWARE PERFORMANCE

The x86 instruction architecture provides an extensive set of functions, but also has many well-known architectural deficiencies: two-operand instructions, inadequate number of registers, the condition code architecture, and so forth. The alternate instruction set eliminates many of these x86 deficiencies thus potentially providing for improved software performance.

This Application Note is intended for sophisticated programmers who may be able to utilize the alternate instruction capability to obtain increased software performance.

1.3 PROCESSOR VARIATIONS

There are variations in the implementation of the alternate instruction capability within the VIA C3 processor family. This document describes the alternate instruction capability for the VIA C3 Samuel, Samuel 2, Ezra and C5XL (Nehemiah) processors. In particular, C5XL differs from the others in certain aspects, and these differences are noted in this document with " in italics. C5XL has improved handling of interrupts and exceptions while in alternate instruction execution mode, therefore most of the sections in Chapter 4 apply only to C5XL and are so noted in the section headings. Section 2.1 describes the CPUID functions that can be used to identify the processor version and its capabilities.

1.4 GENERAL WARNINGS & DISCLAIMERS

The alternate instruction capability provides substantial additional function over the x86 instruction set. Part of this additional functionality is the ability to bypass many of the x86 architecture "protection" mechanisms. The rules documented in this manual must be followed to avoid damaging the integrity of other applications or an underlying operating system.

Thus, use of the alternate instruction capability is restricted (by limiting documentation) to those who have a justified need for this additional capability and can demonstrate the technical ability and "maturity" to properly use the alternate instructions.

In addition, since the alternate instructions expose the underlying implementation, some details of the alternate instruction set change between processor versions.

CHAPTER

x86 ARCHITECTURE INTERACTION

This chapter describes how to enable, invoke, and terminate the alternate instruction mode. The effect on x86 instructions when in alternate instruction mode is described.

2.1 CPUID IDENTIFICATION

System software should use the CPUID instruction to identify the processor and its capabilities. Do not assume that future processors in the VIA processor family will implement the Alternate Instruction Set or that it will be implemented in a backward-compatible manner. In order to determine if the processor supports the Alternate Instruction Set, system software should follow the following procedure.

Identify the processor as a member of the VIA processor family by checking for a Vendor Identification String of "CentaurHauls" using CPUID with EAX=0. Once this has been verified, system software must determine the processor version in order to properly use the Alternate Instruction Set.

In general system software can determine the processor version by comparing the Family and Model Identification fields returned in EAX by the CPUID standard function EAX=1.

	31:14	13:12	11:8	7:4	3:0
EAX	Reserved	Type ID	Family ID	Model ID	Stepping ID
	18	2	4	4	4

The specific values for the VIA C3 processors described in this document are:

PROCESSOR	TYPE ID	FAMILY ID	MODEL ID	STEPPING ID
VIA C3 Samuel (C5A)	0	6	6	Varies
VIA C3 Samuel 2 (C5B)	0	6	7	0-7
VIA C3 Ezra (C5C)	0	6	7	Begins at 8
VIA C3 Nehemiah (C5XL)	0	6	9	Varies

If the processor version is not recognized then system software must not attempt to enable or use the Alternate Instruction Set.

The VIA C3 Nehemiah (C5XL) processor supports Centaur Extended CPUID Functions that should be used to determine if the processor supports Alternate Instruction Set and whether it is enabled. Extended CPUID functions are requested by executing CPUID with EAX set to 0xC0000000 or 0xC0000001.

The following table summarizes the Centaur Extended CPUID Functions.

Centaur Extended CPUID Functions (VIA C3 Nehemiah)

EAX	TITLE	OUTPUT
C0000000	Largest Centaur Extended Function Input Value	EAX=C0000001 EBX,ECX,EDX=Reserved
C0000001	Centaur Extended Feature Flags	EAX,EBX,ECX=Reserved EDX=Centaur Extended Feature Flags

Largest Centaur Extended Function Input Value (EAX==0xC0000000)

VIA C3 Nehemiah (C5XL) returns 0xC0000001 in EAX, the largest Centaur extended function input value. Note that VIA C3 Samuel, Samuel 2, and Ezra do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or EAX=0x00000000 when CPUID is executed with EAX=0xC0000000.

Centaur Extended Feature Flags (EAX==0xC0000001)

Returns Centaur extended feature flags in EDX, these correspond to some Centaur unique features, like the Alternate Instruction Set:

EDX[0]=0 No AIS Support, FCR[0] is zero and WRMSR cannot set it.

1 AIS supported, WRMSR can modify FCR[0].

EDX[1]=0 AIS is Disabled, JMPAI EAX (opcode 0F 3F) will cause Invalid Opcode Exception (INT6).

1 AIS is Enabled, JMPAI EAX (opcode 0F 3F) will enter Alternate Instruction Execution Mode.

EDX[31:2] Reserved

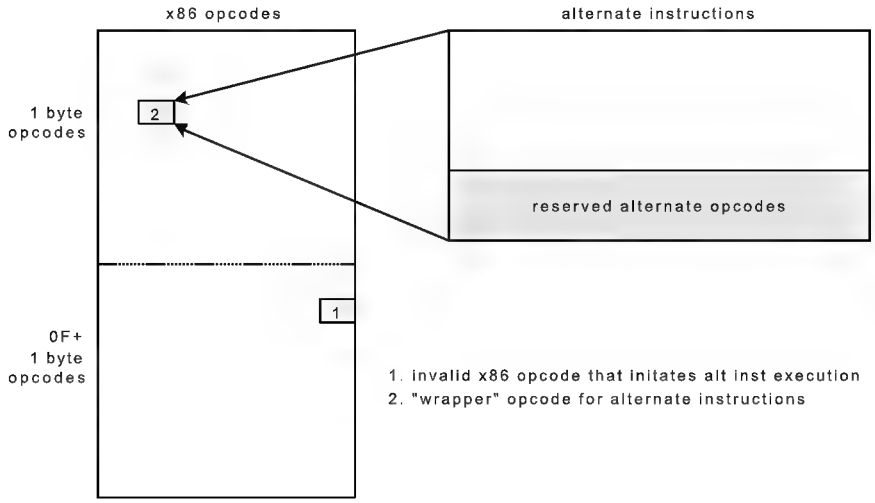
2.2 USAGE MODES

The VIA C3 processor has three major modes of operation:

- **x86 execution mode.** The normal execution mode as defined by the VIA C3 processor datasheets.
- **Alternate instructions enabled mode.** This is the same as x86 mode except that one x86-mode opcode that is normally invalid is now a valid instruction. This new instruction (JMPAI EAX), when executed, switches execution to alternate instruction execution mode. The execution of other x86 instructions is the same as in x86 mode.

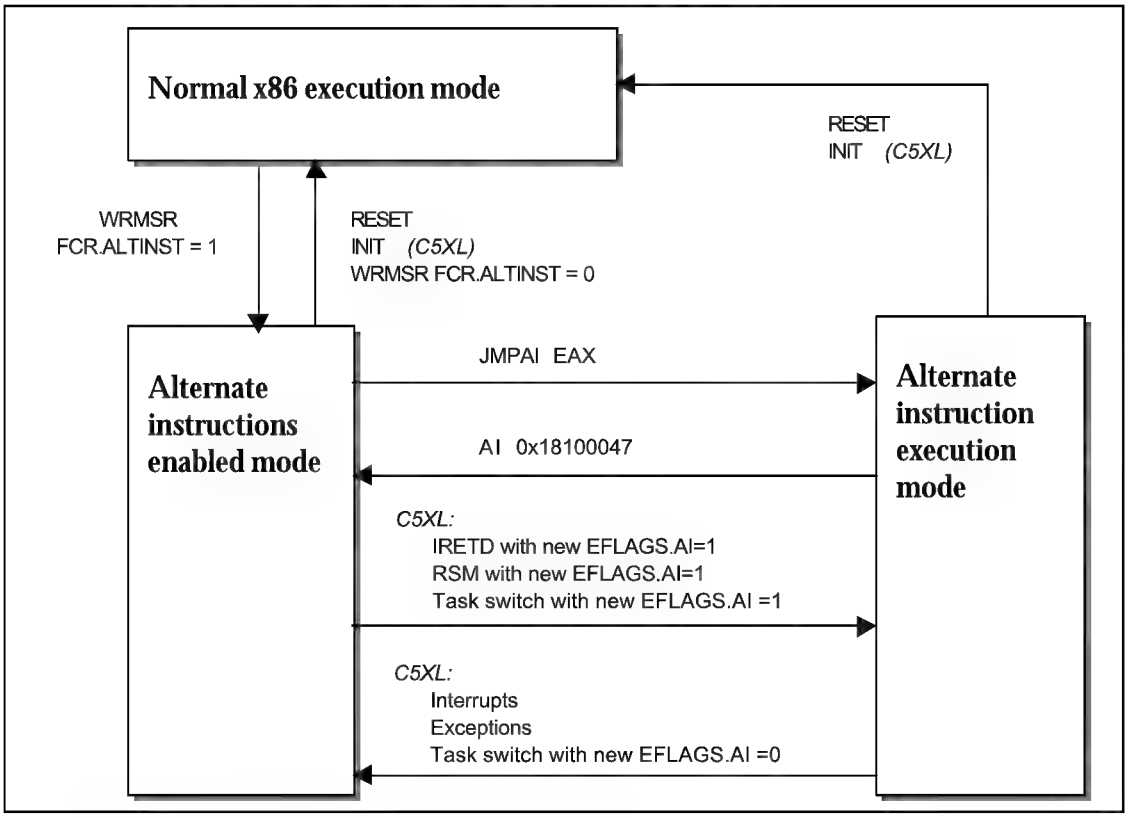
  In Figure 1 normally invalid x86 opcode "1" (JMPAI EAX) becomes a branch to alternate instruction execution mode when in alternate instruction enabled mode.
- **Alternate instruction execution mode.** This mode is entered from alternate instructions enabled mode by execution of the new x86 branch instruction, JMPAI EAX. Following execution of this instruction, a particular x86 instruction is transformed so that it becomes a "wrapper" instruction that carries a micro-operation as its 32-bit displacement. When an alternate instruction is fetched and decoded in alternate instruction execution mode the "wrapper" portion of the alternate instruction is discarded, and the following micro-operation is queued for the execution units to process. All other x86 instructions work as before.

Figure 1. Opcode Mapping



In Figure 1, normally valid x86 opcode "2" is replaced, in alternate instruction execution mode, by a complete set of alternate instructions. The remaining x86 instructions all work normally (but some have restricted use for functional reasons).

Figure 2. Alternate Instruction Mode Transitions



2.3 ALTERNATE INSTRUCTIONS ENABLED MODE

The ability to use alternate instruction mode is controlled by a bit in the VIA C3 processor FCR MSR at address 0x1107. When the ALTINST bit (bit 0) is set to 1, execution of a new x86 instruction is enabled. This setting must be done using a read-modify-write sequence to preserve the values of other FCR bits.

On processors other than C5XL, there be (depending on processor version and stepping) other functional sideffects of setting the ALTINST bit

The new x86 instruction enabled is

Opcode	Instruction	Description
0F 3F	JMPAI EAX	Near jump to address in EAX and enter alternate instruction mode

If FCR.ALTINST is 0, this is an invalid opcode. While setting an FCR bit is a privileged operation, starting alternate instruction mode by executing JMPAI can be done from any protection level.

2.4 ALTERNATE INSTRUCTION EXECUTION MODE

Once ALTINST bit in the FCR MSR has been set to 1, the mechanism for initiating execution of alternate instructions mode is as follows:

1. The ALTINST bit enables execution of the JMPAI instruction that starts execution of alternate instructions. This new branch instruction can be executed from any privilege level at any time that ALTINST is 1. The JMPAI instruction is a two-byte instruction: 0x0F 0x3F. If ALTINST is 0, the execution of JMPAI causes an Invalid Instruction exception.
2. When executed, the new JMPAI x86 instruction causes a near branch to the value in EAX. That is, the branch function is the same as the existing x86 instruction

   jmp eax

   In addition to the branch, the JMPAI instruction sets the processor into an internal mode where the target bytes are not interpreted as x86 instructions but rather as alternate instructions.

   > In alternate instruction execution mode, bit 31 of EFLAGS is set to 1. This bit is reserved in the x86 architecture and is 0 in x86 execution mode and alternate instruction enabled mode in VIA C3 processors. This EFLAGS bit allows transparent management of alternate instruction mode across exceptions and interrupts (see Chapter 4).

3. Following the JMPAI EAX branch, the instructions fetched are treated as one of two types:
   - Normal x86 instructions; these are all normally executed x86 instructions except for the second category of instructions. Some of these x86 instructions, however, should not be used in alternate instruction execution mode (see Section 2.6).
   - x86 instruction opcodes that are used as a wrapper for alternate instructions

Opcode (Wrapper)	**Instruction**	**Description**
8D 84 00	AI **uop32**	Alternate instruction with micro-operation **uop32**

The alternate instructions have the following memory format:

0xYY...XXXXXXXX

where 0xYY... is the "wrapper" opcode for the alternate instruction, and XXXXXXXX is the 32-bit micro-operation contained in the x86 displacement field of the wrapper instruction stored little-endian order. For example, the 32-bit micro-operation to return control from alternate instruction execution mode is 0x18100047, where 0x18 is

the most significant byte and 0x47 is the least significant byte. The mnemonic for this alternate instruction would be written as

AI 0x18100047

This would be encoded for an Ezra processor as follows:

8D 84 00 47 00 10 18

Notice that the four bytes for the micro-operation are stored in little-endian order, consistent with 32-bit displacements in x86 architecture. The least significant byte (0x47) is at the lower address and the most significant byte (0x18) is at the highest address.

For C5XL, the encoding of AI 0x18100047 is:

62 80 47 00 10 18

Details of the micro-operations and registers that may be used in alternate instruction execution mode are provided in the , which is also a restricted document.

4. Each processor version has a potentially different wrapper prefix. For Samuel, Samuel 2, and Ezra, the wrapper is 0x8D 0x84 0x00. That is, the alternate instructions are presented as the 32-bit displacement of a

   LEA [EAX+EAX+disp32]

   instruction.

For , the alternate instruction wrapper is 0x62 0x80. That is, the alternate instructions are presented as the 32-bit displacement of a

BOUND [EAX+disp32]

instruction.

These examples assume that the address size is 32-bits, if it is 16-bits, then an address size prefix (0x67) must be placed in front of the wrapper opcode.

No form of the particular x86 opcode used for the alternate instruction wrapper can be used in alternate instruction mode: for example, all forms of LEA instructions (for C5A, C5B, C5C), or all forms of the BOUND instruction (for C5XL) are restricted.

5. Upon fetching, the wrapper bytes are stripped off and the 32-bit micro-operation contained in the displacement field is directly passed to the execution unit.

6. The VIA C3 Samuel, Samuel 2 and Ezra processors (not C5XL) require that the following instruction be the first instruction executed in Alternate Instruction Execution Mode (following the JMPAI EAX instruction):

   AI 0x83E00819

2.5 TERMINATING ALTERNATE INSTRUCTION EXECUTION MODE

The alternate instruction set contains a special branch instruction that explicitly returns control to alternate instruction enabled mode. The x86 state upon return, however, is not necessarily what it was when alternate instruction execution is entered since the alternate instructions can completely modify the x86 state.

The instruction to return to alternate instruction enabled mode is

AI 0x18100047

This performs a branch to the address in EAX. The EFLAGS bit 31 is set to 0. Alternate instruction mode is still enabled, but the x86 wrapper instruction (LEA or BOUND) now performs as defined in x86 mode.

2.6 X86 ARCHITECTURE RESTRICTIONS

When in alternate instruction execution mode, all x86 instructions can also be used with normally expected behavior for a specific set of x86 instructions. Some common restrictions exist across all processors:

- The x86 IRET and RSM instructions, and any instruction that causes a task switch from protected mode to virtual 86 mode should not be used.
- There are many x86 instructions that can be used that destroy the values in some of the additional alternate instructions registers. This means that the programmer must choose between using these additional registers, and using certain x86 instructions. Appendix A (available in a later version of this document) will contain the detailed list of instruction and which additional registers they use.

In addition to specific x86 instructions that do not work, or should not be used, there are some limitations to execution modes and other pervasive x86 architecture features:

- x86 debug features (breakpoints, single-step mode, etc.) do not work as expected—don't even dream of using these things on alternate instructions.

> C5XL handles interrupts and exceptions differently, so it is possible to use x86 debug features in alternate instruction execution mode in C5XL.

CHAPTER

3

ALTERNATE INSTRUCTION FEATURES

This chapter briefly summarizes the characteristics of the VIA C3 processor alternate instruction set. The instruction set details are provided in another document.

3.1 GENERAL INSTRUCTION FEATURES

Alternate instructions have the following general characteristics:

- All alternate instructions are 32 bits in length, but are contained within a six- or seven-byte x86 instruction "wrapper".
- Most alternate instructions reference two source registers and a destination register .
- In most instructions, one of the source registers can be replaced with a small (encoded) immediate operand
- Some instructions have a 16-bit immediate source operand.
- All instructions that can set the conditions in EFLAGS have an instruction field that suppresses EFLAGS modification.
- Most instructions can explicitly control the operand size to be 32 bits, 16 bits (low), or eight bits (high or low portion of a 32-bit register). Some instructions can also operate on the high 16 bits of a 32-bit register.
- All instructions are either register-register, or they are a load or a store instruction: there are no composite load-ALU type alternate instructions. The load and store instructions, however, can auto-increment and decrement a base address register as a side-effect.

3.2 REGISTERS

The alternate instruction set contains many more registers than the x86 instruction set. Since these registers are used for executing x86 instructions, however, there are potential restrictions on using these additional registers as defined in Chapter 4 and Chapter 5. Following is a summary of the various registers that be usable by alternate instructions:

- **General Registers.** There are 31 total 32-bit general registers (plus a zero-value register). These include the eight x86 general registers, the six x86 selector registers, and a combined LSTR/TR register, leaving 16 additional registers. These additional registers are used by some x86 instructions, thus they may not be available for alternate instruction usage depending on the mix of x86 and alternate instructions.
- **x87 Floating-point Registers.** There are 18 total 80-bit x87 floating-point registers including the eight architected x87 floating-point registers, leaving 10 extended x87 floating-point registers. These additional registers are used by some x87 floating-point instructions, thus they may not be available for alternate instruction usage depending on the mix of x87 and alternate instructions. An advantage to alternate instructions is they can directly access any x87 floating-point register (flat register addressing) in addition to being able to use stack-based floating-point addressing on the eight architected x87 floating-point registers
- **MMX/3DNow! Registers.** The physical registers used by the MMX and 3DNow! instructions are distinct from the x87 floating-point registers and the MMX/3Dnow! registers can be used without affecting the x87 floating-point registers. (Note that C5XL does not implement 3DNow! instructions). There are 10 (16 on C5XL) total MMX/3Dnow! registers including the eight x86 MMX registers, leaving tow (eight) additional MMX registers. These additional registers are used by some x86 instructions, thus they may not be available for alternate instruction usage.

  The advantage of being able to distinctly use the MMX and x87 floating-point registers may be a disadvantage in some cases; the automatic synchronization provided by x86 execution of the two sets of register does not happen when using alternate instructions.
- **Control Registers.** There are many control registers in the hardware implementation. Only two of these may be used by alternate instructions: the architected x86 EFLAGS and CR0 registers. The instructions provided can, however, affect other control registers. Changing any other control register, however, will surely have disastrous and consequences.

3.3 SOME INTERESTING INSTRUCTIONS

Much of the additional capability of alternate instructions comes from general instruction features such as three-operand addressing and the additional registers. The alternate instruction set also includes many useful instructions that have no direct x86 counterpart. Some of these are:

- PUSH & POP-type instructions that can use any general register as a “stack” pointer (as well as any selector register)
- A load instruction that can load two different 32-bit general registers using one 64-bit memory load.

- An instruction that can directly push the IP (avoiding having to CALL to get the IP)
- Instructions that can move data between general registers and MMX/3DNow! registers, between floating-point and MMX/3DNow! registers, and between floating-point and general registers,
- x87 floating-point instructions that can directly access any x87 floating-point register (in addition to being able to use the x87 stack-based floating-point addressing).
- Instructions that AND and OR a value with the EFLAGS register, thus providing fast and direct manipulation of all EFLAGS bits.
- Instructions that can move to and from control registers from any privilege level.

3.4 PROTECTION MECHANISMS

The alternate instruction set supports most of the x86 "protection" mechanisms, but many of these mechanisms are optional and can be bypassed. The following is a summary of x86 protection mechanisms that may be used, or may be bypassed, in alternate instructions:

- Normal x86-style limit and protection checking is performed when the x86 descriptor registers are used. These checks can be bypassed, however, by use of a new "flat descriptor" that is available to alternate instructions.
- Selected x86 control registers such as CR0 can be loaded or stored without any protection checking.

CHAPTER

EXCEPTIONS & INTERRUPTS

This chapter describes the considerations, rules, and alternatives relative to exceptions and interrupts while in alternate instruction mode.

4.1 INTERRUPTS & EXCEPTIONS IN SAMUEL, SAMUEL 2, AND EZRA

VIA C3 processors prior to C5XL (Samuel, Samuel 2 and Ezra) do not support the EFLAGS.AI flag which indicates that a process is in alternate instruction execution mode. These processors will transition out of alternate instruction execution mode only by RESET or execution of the alternate instruction "AI 0x18100047." Therefore, alternate instruction execution mode should be avoided in multitasking operating systems in these processors, or only used with interrupts disabled.

4.2 C5XL: TRANSPARENT INTERRUPTS & EXCEPTIONS

In the C5XL processor an application that uses alternate instructions can be interrupted without interfering with execution of tasks and operating system features that are not aware of alternate instruction mode. In addition, when control is returned to the interrupted application, alternate instruction mode is automatically restored without involvement of the operating system. Certain alternate instruction resources being used by the program, however, may be trashed by the exception handler (this is discussed more in subsequent sections).

The mechanisms providing this transparent use of alternate instructions across interrupts are:

1. When alternate instruction execution mode is entered (using the JMPAI EAX instruction), bit 31 of the EFLAGS register, the is set to 1. This bit is reserved in normal x86 mode and is always zero in x86 mode.

2. The POPFD instruction does not modify the ALTINST bit in EFLAGS. The PUSHFD instruction always stores '0' in the ALTINST bit position of EFLAGS in memory, even if the ALTINST bit is '1' in the EFLAGS register.
3. When an exception or interrupt occurs, the EFLAGS value containing the set ALTINST bit is pushed by the standard x86 interrupt mechanism. We assume that this reserved bit on the stack is not clobbered by the operating system.
4. If the ALTINST bit is set, the processor switches hardware execution to alternate instructions enabled mode and clears ALTINST bit before invoking the x86 exception handler. Thus, exception handlers are executed in alternate instructions enabled mode (or normal x86 execution mode if ALTINST bit was clear);
5. An IRET or RSM instruction restores EFLAGS from a saved memory location. If the ALTMODE bit is set in the restored EFLAGS value, the processor switches execution to alternate instruction execution mode before performing the IRET or RSM branch. Thus,
6. The ALTINST bit in EFLAGS is preserved across task switches. It is saved along with EFLAGS in the TSS of the outgoing task and replaced with the value from the TSS of the incoming task. Thus,

4.3 C5XL: PRESERVING REGISTERS

Although alternate instructions in C5XL can be intermixed with x86 instructions without impacting tasks or the operating system, an exception or interrupt still has an undesirable effect on the task using alternate instructions. The reason is that the contents of additional registers —one of the most useful additional features of the alternate instruction set—are destroyed by an exception or interrupt.

There are three basic approaches to this problem:

- Don't use additional registers with alternate instructions. This reduces the utility of alternate instructions, but they still offer some advantages over x86 instructions even if no additional registers are used.
- Modify the operating system to save and restore the additional registers. This is, for most people, unfeasible.
- Prevent exceptions and interrupts from occurring while additional registers are in use. This is discussed in the next two on masking interrupts.
- Use a unique VIA C3 processor mechanism that allows an application to detect if registers have been trashed by an exception or interrupt. This approach is discussed in section 4.3.3.

4.3.1 MASKING INTERRUPTS

Alternate instructions can be used to mask and unmask INTR at any privilege level (unlike in x86, where this depends on a combination of the CPL, the IOPL, whether in protected mode and whether in V86 mode). The alternate instruction sequence to mask INTR (corresponding to x86 instruction CLI) is:

```
AI    A007FFC0    //  cfc2        tmp7,EFLAGS
AI    2CE7FDFF    //  andil       tmp7,tmp7,~{IF_MASK}
AI    80E0F819    //  ctc2.32     EFLAGS,tmp7
```

The instruction sequence to unmask INTR (corresponding to x86 instruction STI) is:

```
AI    A007FFC0    //  cfc2        tmp7,EFLAGS
AI    34E70200    //  ori         tmp7,tmp7,IF_MASK
AI    80E0F819    //  ctc2.32     EFLAGS,tmp7
```

When this unmask sequence is executed, an INTR interrupt may interrupt execution immediately following the unmask instruction. This sequence will therefore not prevent interrupts from happening while in AIS execution mode even if this sequence is immediately followed by the alternate instruction that exits AIS execution mode.

C5A, C5B and C5C do not handle external interrupts transparently in AIS execution mode, so it may be desirable to re-enable interrupts and exit AIS execution as an atomic sequence. This can be accomplished by using the STI instruction and exiting AIS execution mode immediately afterward (the STI instructions inhibits interrupts until after completion of the instruction following the STI). However, note that the STI instruction may #GP fault based on x86 privilege rules (CPL, IOPL, etc.).

4.3.2 UNMASKABLE INTERRUPTS

The process of masking INTR may not fully enable use of additional registers since there are other non-maskable interrupts that can occur. These other interrupts are:

- **x86 Exceptions**. These cannot be masked, but they can all be easily avoided by the application code except for a Page Faults which can be avoided (albeit not easily) by pinning pages in memory.
- **NMI (Non-Maskable Interrupt).** Fortunately, no modern PC (running Windows) causes an NMI interrupt. Thus, in most cases the possibility of NMI does not have to be considered.
- **SMI.** Unfortunately, all modern PCs (running Windows) cause SMI interrupts. In theory, these can be suppressed via the chipset programming, but this is privileged and very hard to do.
- **Internal bus interrupts.** Some bus signals actually cause interrupts to the microcode. These are transparent at the instruction interface, but the internal interrupt processing uses all of the general alternate instructions registers. Thus, an application cannot use additional general registers unless it can guarantee that the following bus interrupts will not occur:
  - STPCLK
  - FLUSH

These bus interrupts can be suppressed via the chipset programming, but this is privileged and very hard to do, so guaranteeing that these bus interrupts won't occur is not realistic.

4.3.3 C5XL: REGISTER TRASHING DETECTION

The previous section identified the difficulty (impossibility?) of guaranteeing that no exception or interrupt can occur during use of alternate instructions. A mechanism exists, however, that allows code to detect that an exception or interrupt has occurred. This enables the technique of writing a block of code such that it is tolerant of its registers being trashed.

The specifics steps using this approach are:

1. At the start of a block of alternate instructions that use additional registers, a zero value should be written into general register 31.
2. The code using additional registers must not change permanent application state. That is, the code must be written such that it can be re-executed multiple times using the input state and still get the correct result.
3. Upon completing a block of code that used additional registers, register 31 should be tested:
   - If its value is zero, that means that no exception or interrupt occurred and thus no additional register was transparently thrashed. In this case, the results of the code block can be committed.
   - If the register 31 value is not zero, this means that the register values cannot be trusted and the results should not be committed. In this case, register 31 should be

reset to zero, and the code re-executed using the original input values. This test and re-execute process should continue until register 31 indicates that no exception or interrupt occurred to trash the additional register content.

CHAPTER

5

PERFORMANCE

This chapter summarizes basic performance factors for use of alternate instructions. Additional detail about slips and stalls is provided in the detailed alternate instruction set defintion Application Note.

5.1 MODE TRANSITIONS

The JMPAI branch instruction (0x0F 0x3F) that enters alternate instruction execution mode, and the branch that returns to normal x86 instruction mode, are not predicted. In addition, the branch to enter alternate instruction execution mode is trapped to microcode. Thus, these branches take many clocks to execute.

5.2 ALTERNATE INSTRUCTION EXECUTION

The basic execution time for all documented alternate instructions is one clock. These instructions, however, are subject to the same types of slips and stalls that apply to x86 instructions (as documented in xxx).

APPENDIX

A

REGISTER USAGE

A.1 GENERAL INSTRUCTIONS

This Appendix will be provided in a later version of this document.

A.2 X87 FLOATING POINT INSTRUCTIONS